Why Use Concurrency?

- Enable programs that interact with a (naturally concurrent) environment
  - E.g., GUI programming, OS interaction
- Support organization into parts that execute independently and interact as needed
  - E.g., client/server and producer/consumer programs
- Enable speedup by parallel execution

Concurrent Environments

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Number of machines (memories)</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>Standard uniprocessor</td>
</tr>
<tr>
<td>Many</td>
<td>Shared memory multiprocessor</td>
</tr>
<tr>
<td></td>
<td>Many shared memory multiprocessors</td>
</tr>
</tbody>
</table>
Concurrent Models

- Dataflow
- Multiple concurrent sequential programs
- Many others

Dataflow Execution

- Execute all computations that are ready
  - Do not use a variable before it is ‘bound’
  - A statement using the variable suspends until another thread binds the variable
- Avoids complexities associated with other forms of concurrency
- But overhead of naïve implementation is typically too large for practical use
- Well described by Roy and Haridi

Multiple concurrent programs

- Processes with message passing
  - Communicating Sequential Processes (CSP) — Hoare
- Threads with shared memory

Nondeterminism of Concurrent Computations

- Results may depend on order in which shared data accessed
  - Observable nondeterminism: race condition
- Programs with state and concurrency are difficult to reason about and make correct
  - E.g., Therac-25 radiation therapy machines
    - Overdoses by factors of 1000
    - Killed a handful of people
- Difficult to test
  - ‘Heisenbug’
  - But see ConTest
Therac-25

- Cancer treatment machine that emitted photons and electrons
- Accidental overdoses killed 2 and injured 4
- Software allowed concurrent access to shared memory
  - No synchronization
  - Test and set not indivisible
- Some injuries caused by interaction between treatment code and input
  - See pp. 22 – 28 of Leveson
  - See http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Figure.2.GIF

Techniques for Controlling Nondeterminism

- Atomicity
- Locking
- Message passing
- Many others …

Other Undesirable Behavior

- Deadlock
- Starvation

Concurrency in Java

- Threads within a JVM
- RMI
**java.lang.Thread Review**

- JVM runs multiple threads concurrently
- Higher priority threads execute preferentially
- JVM terminates when
  - `Runtime.exit` is called
  - All threads that are not daemon threads die

**Thread Scheduling**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>start()</code></td>
<td>This thread begins execution; the JVM calls the thread's <code>run()</code> method</td>
</tr>
<tr>
<td><code>yield()</code></td>
<td>The currently executing thread temporarily pauses to allow other threads to execute.</td>
</tr>
<tr>
<td><code>sleep(long millis)</code></td>
<td>The currently executing thread temporarily ceases execution for the specified number of milliseconds.</td>
</tr>
<tr>
<td><code>interrupt()</code></td>
<td>Interrupts this thread.</td>
</tr>
<tr>
<td><code>join(long millis)</code></td>
<td>Waits at most millis milliseconds for this thread to die.</td>
</tr>
</tbody>
</table>

**Deprecated Thread Scheduling Methods**

- All these methods were a **bad** idea!
- `resume()`
- `stop()`
- `suspend()`
- Those Java guys really screwed up

**Thread Monitor Actions**

- Each object maintains 1 lock, or monitor

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wait(long timeout)</code></td>
<td>Causes current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed. The current thread must own this object's monitor.</td>
</tr>
<tr>
<td><code>void notify()</code></td>
<td>Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the wait methods.</td>
</tr>
<tr>
<td><code>void notifyAll()</code></td>
<td>Wakes up all threads that are waiting on this object's monitor.</td>
</tr>
</tbody>
</table>
JVM Threads Model

• Synchronizing threads
• Monitor
  – At most one thread at a time can execute a protected region of code
• Monitors are implemented with locks
• One lock per object

synchronized statement
  – Identify the object
  – Acquire (lock) the object’s lock
  – Execute the synchronized code
  – Release (unlock) the object’s lock

Java Memory Model

• Totally ordered actions
  – Actions by a thread
  – Actions by main memory on any one variable
  – Actions by main memory on any lock

References

• Tony Hoare, “Communicating Sequential Processes”, 1985

The End